



Kumori PaaS Quick Start

Kumori Systems v1.0.1, July 2018

Table of Contents

1. Introducing Kumori	1
1.1. The Kumori PaaS Service Model.....	1
1.1.1. An example	1
2. Setting up tools and environment.....	2
2.1. Environment requisites.....	2
2.2. Kumori CLI install.....	3
2.3. Kumori Dashboard.....	3
2.4. Create a workspace.....	3
2.5. Set your workspace domain.....	4
2.6. Obtain your API access token	4
3. Get your first service up and running in Kumori PaaS (Hello World example)	6
3.1. Populate your workspace with Hello World sample service	6
3.1.1. The helloworld component.....	7
3.1.2. The helloworld service.....	8
3.1.3. The helloworld deployment manifest	8
3.2. Build FE component	8
3.3. Register FE component.....	9
3.4. Register Hello World service application.....	9
3.5. Configure Hello World service application	9
3.5.1. Configure logging system (set logzioToken parameter)	9
3.6. Deploy Hello World service.....	10
3.7. Update Hello World service	13
3.7.1. Update and register a new version of helloworld component.....	14
3.7.2. Update and register a new version of Hello World service	14
3.7.3. Deploy a new service	15
3.7.4. Change the endpoint link to the new service	16
3.7.5. Remove the old service	19
4. The Kumori PaaS Service Model in detail	20
4.1. Component.....	20
4.1.1. Component interface	21
4.2. Service application	22
4.3. Service.....	22
4.4. Manifest versioning.....	23
5. Hello World example manifests in detail.....	23
5.1. FE component	23
5.2. Service application	24
5.3. Service application deployment	25
6. Hello World v2: FE + AsciiConverter (with Load Balancing)	26
6.1. Populate your workspace with Hello World V2 sample service	27
6.1.1. The component helloworld_v2_fe.....	27

6.1.2. The component <code>helloworld_v2_ascii</code>	27
6.1.3. The <code>helloworld_v2</code> service	28
6.1.4. The <code>helloworld</code> deployment manifest.....	28
6.2. Configure logging system (set <code>logzioToken</code> parameter).....	28
6.3. Manifests	28
6.3.1. FE component	28
6.3.2. <code>AsciiConverter</code> component	29
6.3.3. Service application	30
6.3.4. Service application deployment	31
6.4. Deploy Hello World v2 service.....	31
6.5. Testing the service	32
6.6. Scaling a role in a service	33

This document describes the basic elements of Kumori PaaS, and guides the developer ('you') through the implementation and deployment of two example Node.js applications.

For sure, this is **not** a complete manual or a full reference guide of Kumori PaaS. In fact, what we will show you here is only a small part of the full potential of the platform.

1. Introducing Kumori

Kumori is a Platform as a Service (PaaS) that eases the development and deployment of an application, and the management of its lifecycle.

Kumori runs on top of an IaaS, from where it obtains and manages resources in a transparent way to application developer, so you don't have to worry about infrastructure.

Kumori mission is to support the creation of elastic applications and manage their entire life cycle, including automatic auto-scaling based on load situation, fulfilling a SLA and minimizing the incurred cost in the underlying IAAS.

Access to Kumori PaaS is currently done via Kumori Dashboard, Kumori CLI, and the underlying Admission REST API. The two first ways are covered, although not thoroughly, in this guide.

1.1. The Kumori PaaS Service Model

In order for a service to be managed by Kumori PaaS, it must follow a very specific service model. This model is defined by the following elements:

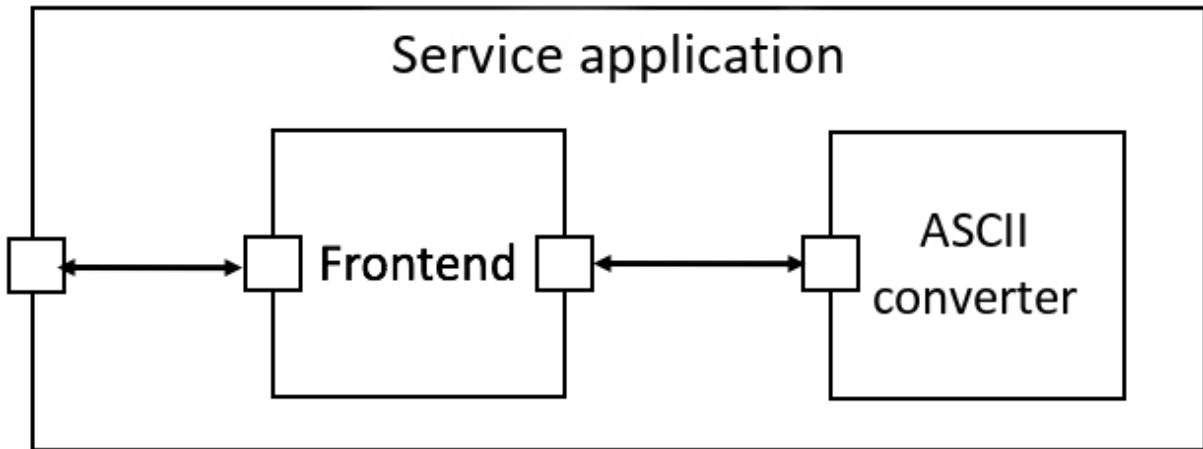
- **Component:** basic unit of execution. It is a runnable and self-contained piece of code that typically implements a certain API. A component can interact with other components through channels, which are provided by the platform, but declared by the component developer. A component can also have some configuration parameters and require some resources to work (e.g. volatile or persistent volumes).
- **Service application:** set of components that work together to provide a final service. Each component can carry out one or more roles. A service application declares a specific topology connecting different role channels through defined connectors.
- **Service:** it is the result of deploying a service application. It is composed by a number of running instances of each role. The number of instances may vary during service lifecycle.

1.1.1. An example

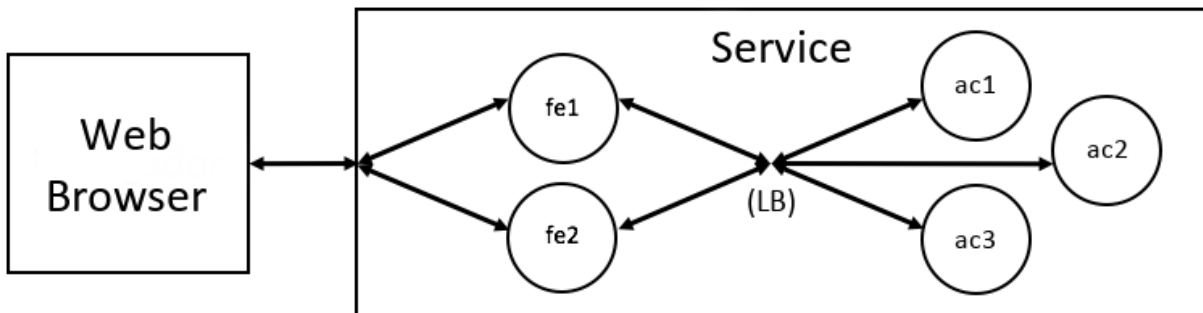
Let's say we want to provide an ASCII art generator as a service that can be accessed through a REST API (this example is covered in detail in section [Hello World v2: FE + AsciiConverter \(with Load Balancing\)](#)). Our service application will be composed by two roles carried out by two components:

- **Asciiart Converter:** converts images to a string of ASCII characters. It requires a single channel through which it handles conversion requests.

- Front End (FE): provides a REST API to interact with Ascii Converter. This component requires two channels. One to attend to REST petitions and the other to issue requests to the Ascii Converter role. This component also servers the service Single Page Application (SPA).



Once deployed, Kumori PaaS will launch several instances of each role. In this example, instances of FE role will be connected to Ascii Converter role instances through channels and a load balancer connector (LB).



2. Setting up tools and environment

2.1. Environment requisites

- Linux OS (current LTS version of Ubuntu or equivalent)
- Docker CE
- Node.js v8
- Npm
- Git
- zip
- curl



As stated in the [Docker installation guide](#), to use Docker as a non-root user, you should add your user to the "docker" group with: `sudo usermod -aG docker $USER`

2.2. Kumori CLI install

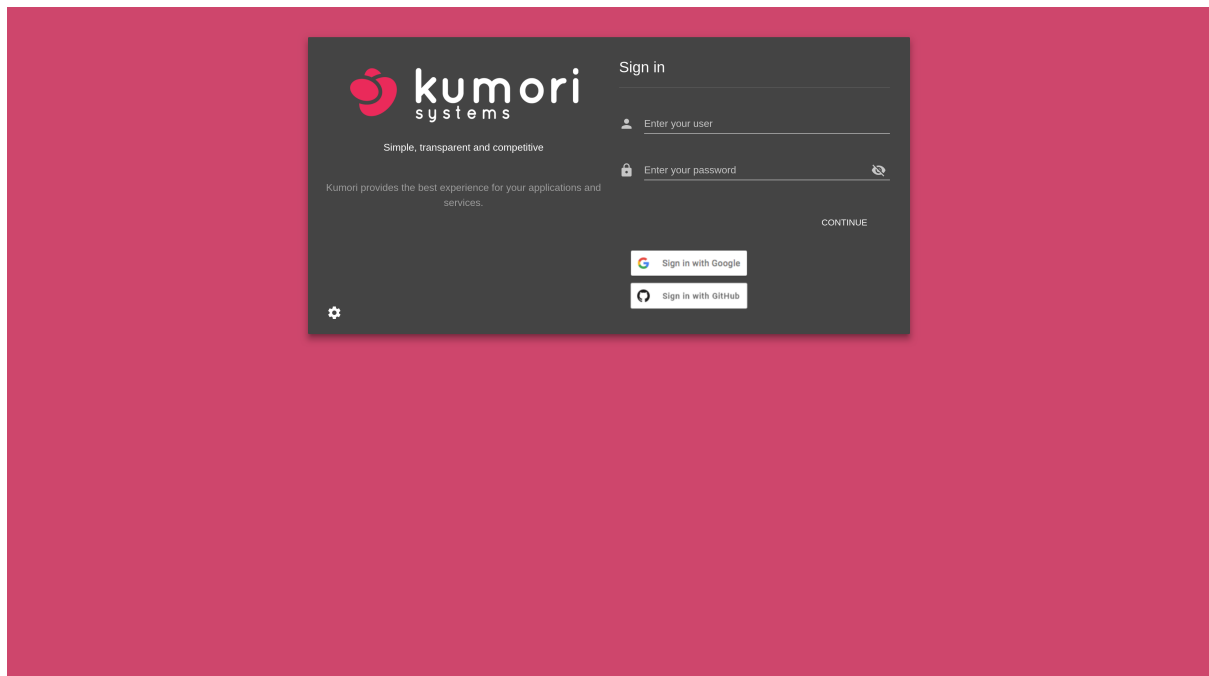
Kumori CLI (command-line interface) is a tool intended to boost the process of developing elements for Kumori PaaS. It is available in [npm](#).

```
$ sudo npm install -g @kumori/cli
```

2.3. Kumori Dashboard

Kumori Dashboard is available in <https://dashboard.baco.kumori.cloud>.

Be sure you already have a Kumori account, because you will need to access Dashboard in further sections. Otherwise, sign up and get a free account [here](#).



2.4. Create a workspace

Kumori CLI uses the concept of workspace, where all the elements related to a project are placed. To create a new workspace, switch to an empty directory and run `kumori init` command:

```
$ mkdir workspace
$ cd workspace
$ kumori init
  create kumoriConfig.json
  create builtts/README.md
  create components/README.md
  create dependencies/README.md
  create deployments/README.md
  create resources/README.md
  create runtimes/README.md
  create services/README.md
  create tests/README.md
```

`kumoriConfig.json` file contains the configuration used in this workspace, including the API access token used to interact with the platform.

2.5. Set your workspace domain

In Kumori PaaS, every element is identified by a unique Uniform Resource Name (URN). This URN is set when the element is registered in the platform and follows the pattern `eslap://<DOMAIN>/<TYPE>/<NAME>/<VERSION>`. In order to avoid collisions with other customers that would prevent you from registering parts of your project, you need to set a domain of your own choice, which is done at workspace level (although it could be overridden when adding particular elements):

```
$ kumori set domain YOUR_CHOSEN_DOMAIN
```

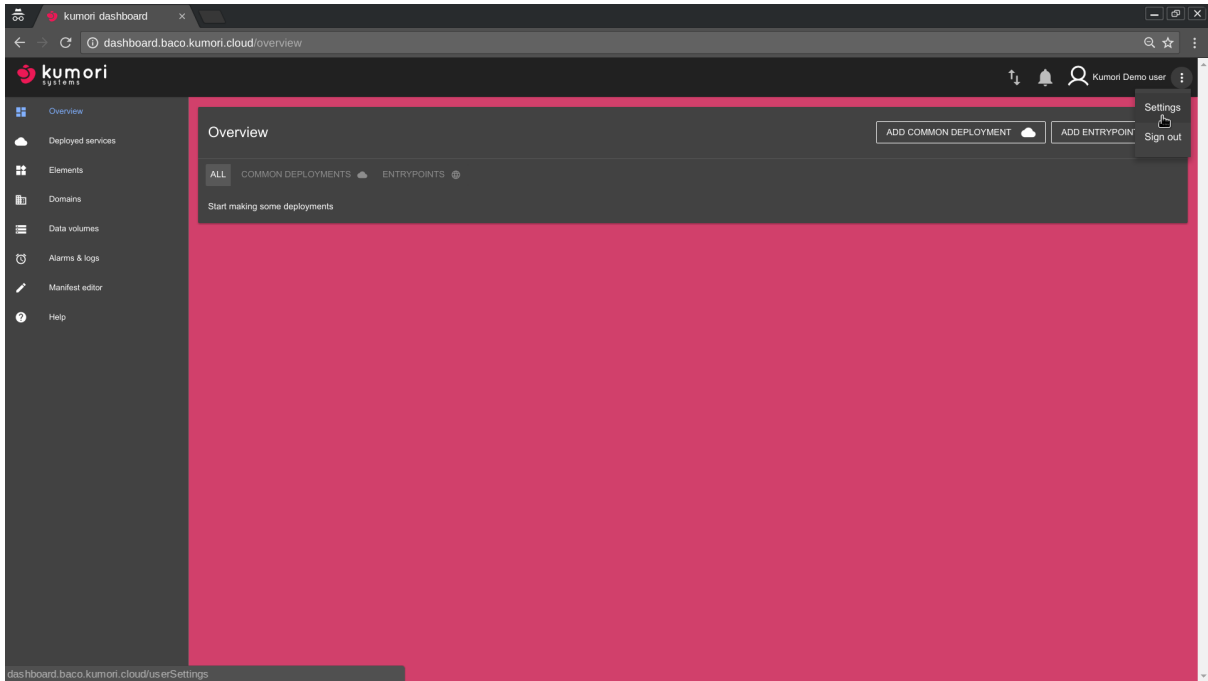


The `<DOMAIN>` is only used to name your elements. Hence, it can be random or invented and you don't have to register it anywhere beside your local workspace. However, we strongly suggest you to use your own company domain to avoid collisions with other company's components.

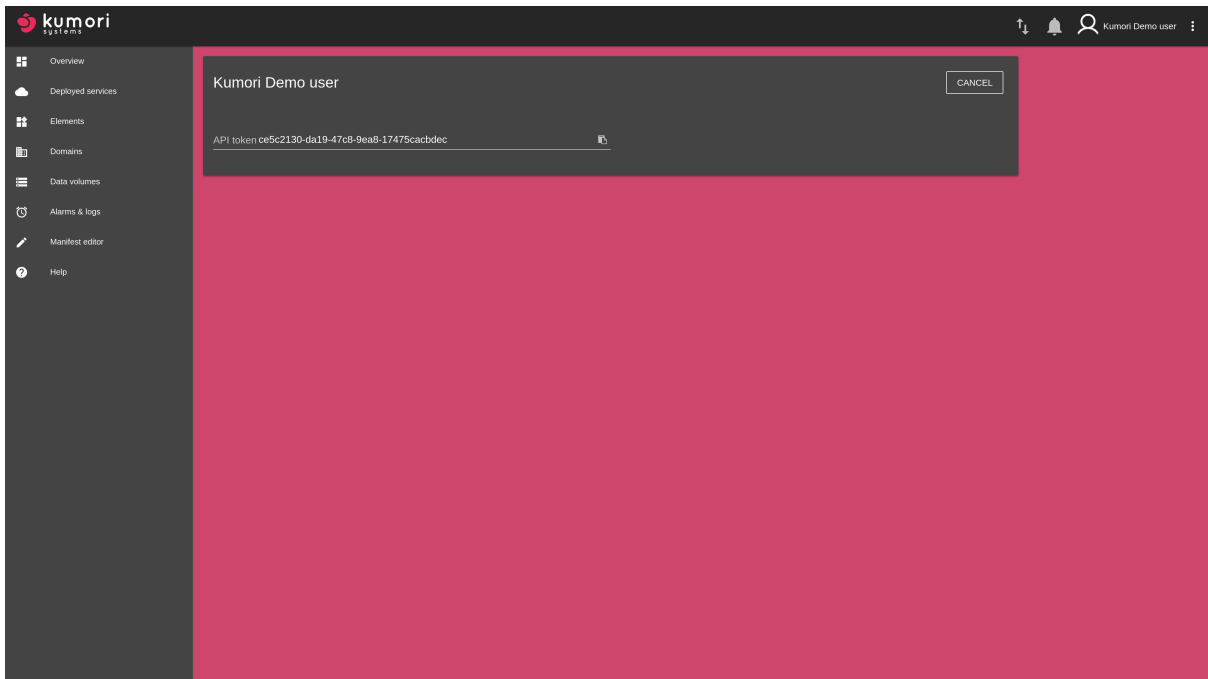
2.6. Obtain your API access token

The API access token is used by Kumori CLI to operate with the platform. To obtain it, sign in [Kumori Dashboard](#) using the same authorization system (Google, GitHub...) you used during sign up. This is important, as the different systems are not linked, even if two accounts share the same email.

Open the three-dot menu at the top right of the page, next to user name, and click on Settings.



The main view will change to the one below:



Copy the API access token.

Then, run the following Kumori CLI commands:

```
$ kumori stamp update -t YOUR_API_ACCESS_TOKEN baco
```



BACO is the current production version of Kumori PaaS and comes preconfigured by default in the workspace.

Now, you will be able to interact with the platform from your workspace.

For security reasons, API access tokens expire after 30 days. In the future, you will be able to issue and revoke permanent tokens in Dashboard, but at the moment, you will need to repeat this operation every month.

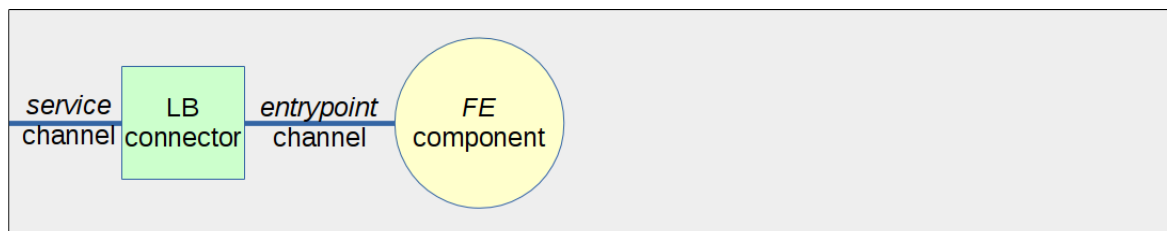
3. Get your first service up and running in Kumori PaaS (Hello World example)

We are going to start with a very simple service application, composed by a single component named FE (Front End), developed using Node.js and Express.

FE serves a static sample page in / (so you can quickly test the service in your browser) as well as exposing a simple REST API in /api/, with two routes: /api/sayhello, which always returns "Hello World!" message, and /api/echo/:msg, which returns the message passed as part of the URL.

Not surprisingly, the topology of this service lacks channels and connectors for interconnecting components.

The component is going to have a single channel that allows to access the service from the outside, through a domain name.



3.1. Populate your workspace with Hello World sample service

First, we need to add the Hello World sample service elements to our workspace by using the `@kumori/workspace:project-hello-world` Yeoman generator. To use it, we have to provide at least a name for the new project and we will use `helloworld`.



Only alphanumeric characters can be used in names. That includes numbers, letters and underscores. So, for example, `helloworld` and `hello_world` are allowed but `hello-world` is forbidden.

```
$ kumori project add -t @kumori/workspace:project-hello-world helloworld
```

This will create:

- The `helloworld` component under `components/YOUR_CHOSEN_DOMAIN/helloworld/workspace` path.

- The `helloworld` service application under `services/YOUR_CHOSEN_DOMAIN/helloworld/`` workspace path.
- The `helloworld` deployment manifest under `deployments/helloworld` workspace path.

3.1.1. The `helloworld` component



The source code of this example is extensively commented and we encourage you to review it for implementation details.

A component is nothing more than a class that inherits from `Component` (more on this later) and implements the necessary methods used by the Kumori runtime (in this example, the one corresponding to Node.js) to manage its lifecycle.

Internally, the `helloworld` component contains several files:

package.json

The usual in a project for Node.js. It is worth noting that it includes dependencies to two Kumori modules:

- `component`: class that our component must extend.
- `http-message`: equivalent to the Node.js `http` module, but that listens on a Kumori channel instead of IP+port.

Currently, those dependencies are git repositories but will be registered soon as npm modules and, most probably, also included in the platform runtime environments used to run the components.

Apart from that, it includes some custom NPM scripts, being `dist`, `superclean` and `devinit` mandatory.

taskfile.js

Contains Taskr configuration and targets. Taskr is a task automation tool, much like Gulp or Grunt, but with some nice features. In our templates, we provide targets for installing dependencies, building and generating distributable versions of the components, the ones that Kumori CLI uses under the hood.

Manifest.json

It is the descriptor of the component. We will explain its content later on in section [Hello World example manifests in detail](#). For the moment, just mentioning that it defines component channels and parameters.

lib/restapi.coffee

REST API implementation. It uses an average Express server but employs Kumori's custom `httpMessage` module instead of using Node.js `http` module.

lib/index.coffee

FE component implementation. Its methods `constructor`, `run`, `shutdown` y `configure` are invoked by Kumori PaaS, managing instance lifecycle. In its `constructor` the RestAPI object is created, which is started in `run` method.

static/

Contains the files used for serving the static sample page, i.e. a bunch of HTML, JS, image and font files.

3.1.2. The `helloworld` service

A service application makes use of components, assigns them roles, defines connectors, defines service channels and declares a specific topology connecting role channels with other role channels or service channels through defined connectors. This is declared in `Manifest.json` descriptor file.

3.1.3. The `helloworld` deployment manifest

A deployment represents the act of running a service application, which creates a service. The new service configuration is provided in the deployment manifest `Manifest.json`. This file and contains:

- The parameters values (if any).
- The resources assigned (if any).
- The roles arrangement (RAM, CPU, instance boundaries, ...).

Note that the file itself does not runs anything. Just contains the configuration needed in the deployment process. The deployment itself will be performen later in section [Deploy Hello World service](#).

See section [The Kumori PaaS Service Model in detail](#) for more information.

3.2. Build FE component

Running a service applications implies running several instances of each involved components. To run a component we need first to build it which means creating a zip file containing everything the platform will need to run it. What should be included in that file depends mostly on the language used to develop the component. In this example, FE is a NodeJS component and to build it we need first to install its dependencies. Whatever the building process does, it ends creating a distrutable file containing everything the platform will need to run a component instance.

We do not recommend to run `npm install` directly, since we may have some dependencies on the operating system libraries (e.g when compiling some module, using `node-gyp` in this case) that may cause troubles later.

Therefore, we suggest to use Kumori CLI, which internally runs some scripts defined in `package.json` that essentially execute `npm install` in the same runtime environment (i.e. same Docker image) needed to run that component instances in the platform. The specific runtime is defined in component manifest, but more on that later in section [Hello World example manifests in detail](#).

To build the component, run the following Kumori CLI command:

```
$ kumori component build helloworld
```



This operation makes use of Docker. In case you haven't added your user to `docker` group, you will need to run the above command with `sudo`.

3.3. Register FE component

Now it's time to register our previously built component in Kumori PaaS.

To do so, run the following Kumori CLI command:

```
$ kumori component register helloworld
```



This requires that you have configured a valid API access token. If that's not the case, please revisit section [Obtain your API access token](#).

3.4. Register Hello World service application

Once the component is registered, the service application must be registered too.

To register it in Kumori PaaS, run the following Kumori CLI command:

```
$ kumori service register helloworld
```

3.5. Configure Hello World service application

As explained in section [Populate your workspace with Hello World sample service](#), during the population step a deployment manifest is created in `deployments/helloworld`.



You can create as many deployment configurations as you want, even for the same service application. Just execute `kumori deployment add` to add more deployment manifests to your workspace.

The `Manifest.json` file, that we will explain later on in section [Hello World example manifests in detail](#), contains all service application deployment configuration. In this example, we can configure the logging system.

3.5.1. Configure logging system (set `logzioToken` parameter)



The following is not mandatory but we encourage you to do it, especially if you introduce modifications in the code.

The FE component can use a logging system. This is especially useful when running software on a managed environment, i.e. when running your service applications in Kumori PaaS. There are many solutions available, we have just chosen the simplest one to our knowledge.

Go to <https://logz.io/freetrial/> and create an account.

You will be immediately logged in, you don't even need to click on any confirmation email.

Then, go to <https://app.logz.io/#/dashboard/settings/general> and copy account token.

Finally, edit `deployments/helloworld/Manifest.json` and set `logzioToken` value.

You will be able to check logs in <https://app.logz.io/#/dashboard/kibana> once you have deployed the service.

3.6. Deploy Hello World service

It's time to deploy! Once again, you can do it using Kumori CLI:

```
$ kumori deployment deploy helloworld
```

As a result, you will get some information about the new service, including its unique name (URN).

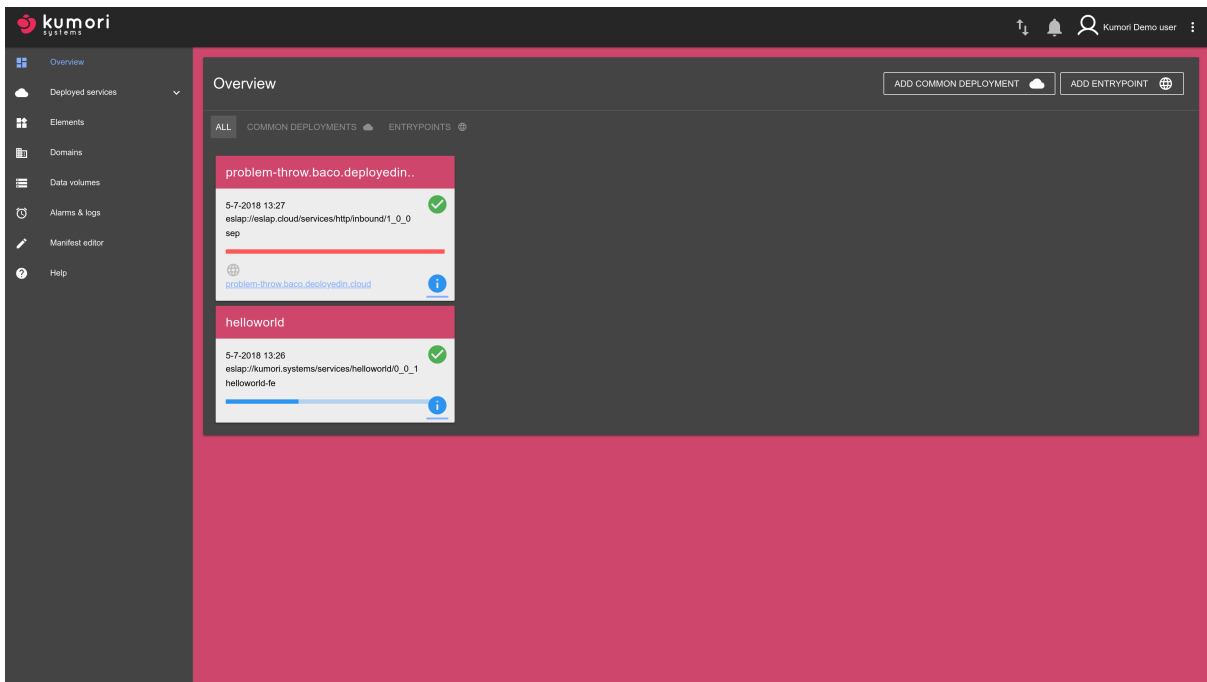
```
-----  
Elements already registered in baco:  
* eslap://kumori.systems/services/helloworld/0_0_1    SKIPPED  
* eslap://kumori.systems/components/helloworld/0_0_1  SKIPPED  
-----  
Service deployed:  
Nickname:  helloworld  
URN:      slap://kumori.systems/deployments/20180702_150513/ed8a7c2e  
Role:     helloworld-fe  
-----
```



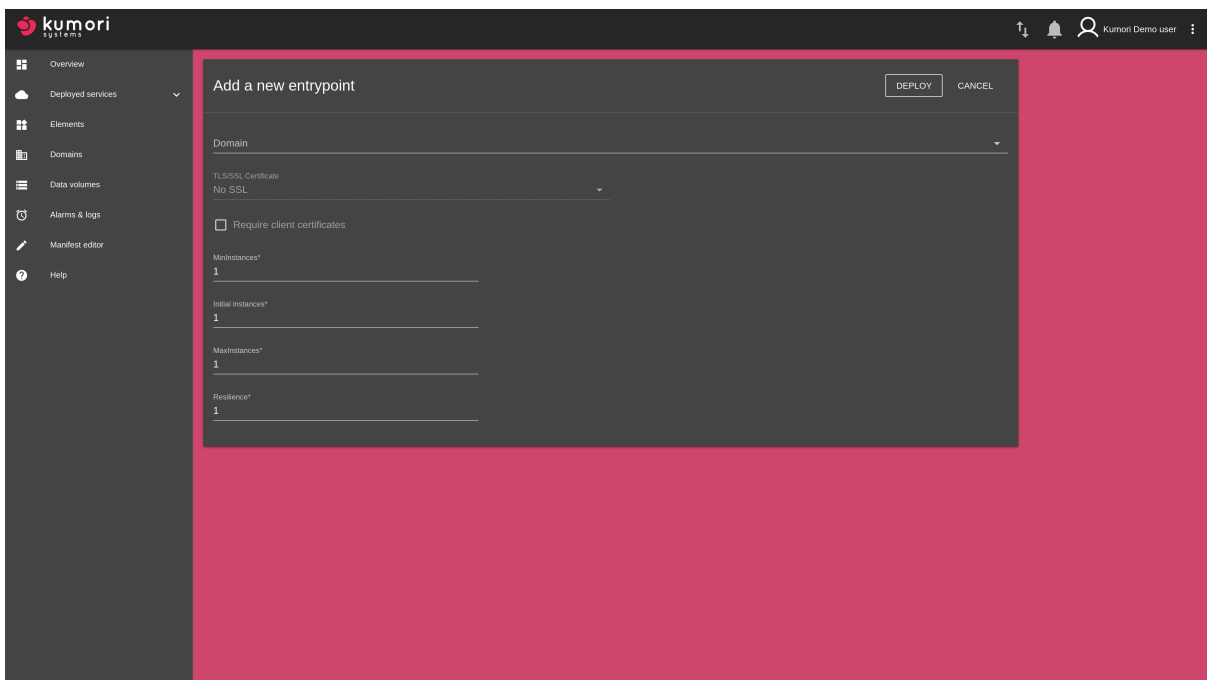
As expected, the output points out that the component and service have been already registered because we registered them earlier. As we will see later in section [Hello World v2: FE + AsciiConverter \(with Load Balancing\)](#), we can build the components, register everything and deploy a service application in a single step. However, for this very first example, we think is far more educational to show all those actions step by step.

Keep the URN since you will need it later.


At this point, you have deployed your service, but it is not accesible from outside. Go to [Kumori Dashboard](#) - Overview, and press Add Entrypoint button.

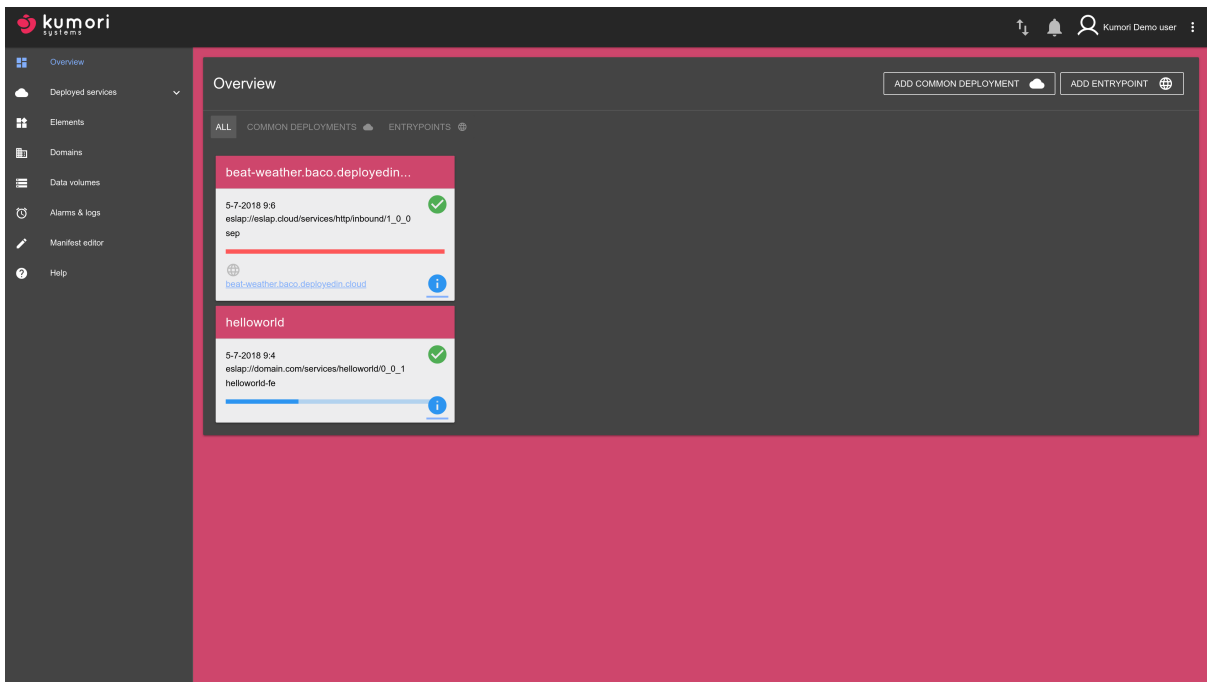


Do not select any domain, so a random domain is generated, and press Deploy button.



After a while, this will deploy an HTTP inbound service.

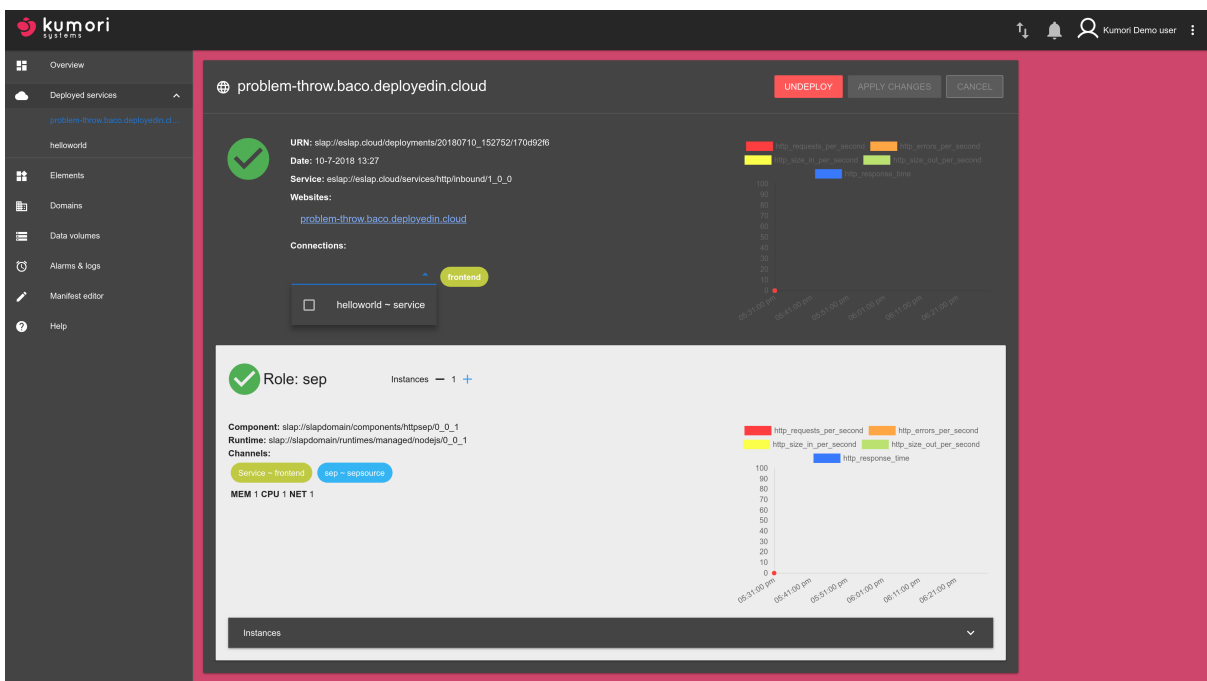
Then, click on the blue Info button  of the newly created HTTP inbound service.



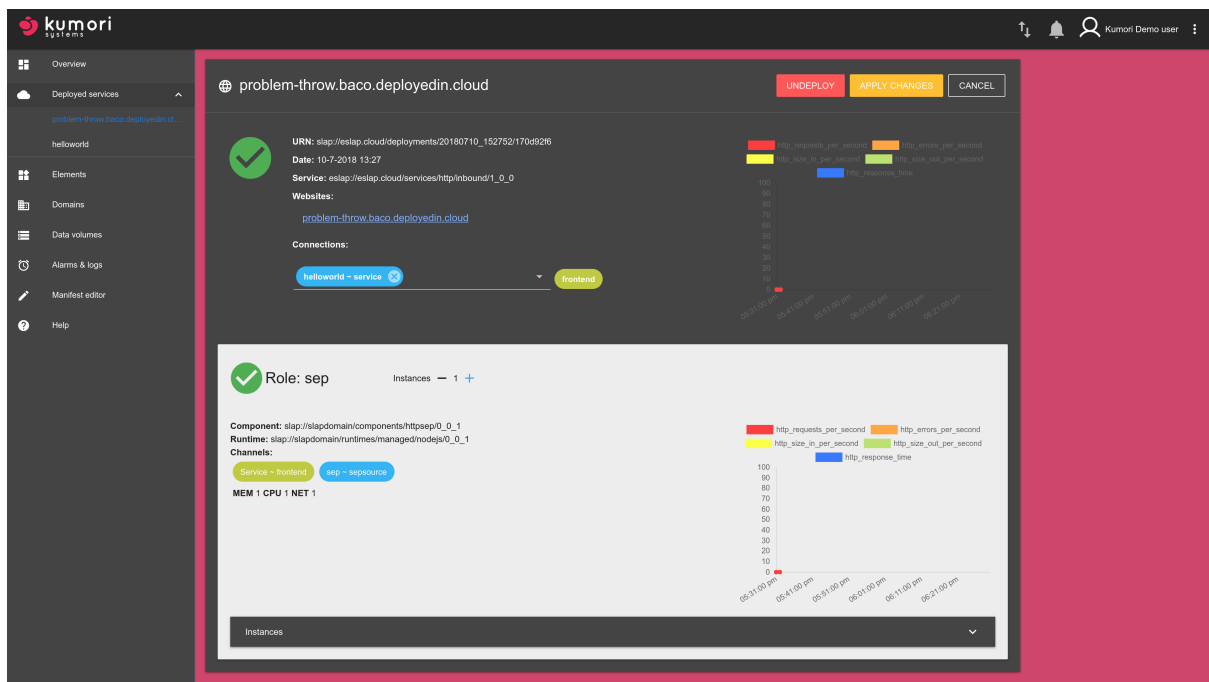
This shows that service in detail, including the assigned random domain. Under *Connections*, on the dropdown menu next to **frontend** (that's a service channel), select **helloworld ~ service**.



You will not be able link the Inbound service until at least one instance of the *helloworld-fe* role is created. If the drop down menu is empty, wait a few minutes.



Press Apply Changes button



This links the HTTP inbound service with the Hello World service through their service channels.

Finally, go the provided website URL and have fun.



If you have been extremely quick doing the above steps and see an error message telling you that the requested service is not deployed, just wait a bit and press Try Again button.



Apart from the static page available in `/`, check `/api/sayhello` and `/api/echo/:msg` routes from your browser or command-line `curl`.

Hooray! You have deployed your first service in Kumori PaaS. Now, let's see what we have done and give some insights. Now let's see how we can update it.

3.7. Update Hello World service

In Kumori Platform, registered elements (like a component or a service) are immutable and cannot be changed. This ensures that an URN will allways point to the same element. If we need to update an element it must be registered with a different URN. The most straightforward way of doing that is increasing the version part of the original element URN (see section [Manifest versioning](#)).

In this section we show how a new version of our *helloworld* service can be created, registered and deployed. This new version will return `Hello world updated!` instead of `Hello world!` when the `/api/sayhello` API call is invoked. We also show how the inbound we created before (see section [Deploy Hello World service](#)) can be linked to the new version of the service to keep the same domain.

3.7.1. Update and register a new version of `helloworld` component

First, we update the `helloworld` component.

Edit the file `components/YOUR_CHOSEN_DOMAIN/helloworld/lib/restapi.js` and change the `/api/sayhello` handler to return `Hello world updated!`

```
_createExpressApp() {
  ...
  app.use('/api/sayhello', (req, res, next) => {
    res.status(200).send('Hello world updated!');
  });
  ...
}
```

Edit the file `components/YOUR_CHOSEN_DOMAIN/helloworld/Manifest.json` and change the component name version to `0_0_2`:

```
{
  "spec": "http://eslap.cloud/manifest/component/1_0_0",
  "name": "eslap://YOUR_CHOSEN_DOMAIN/components/helloworld/0_0_2",
  ...
}
```

Once updated, we build and register the new version as we did with the previous version in sections [Build FE component](#) and [Register FE component](#). Both versions can coexist because they have a different URN.

```
$ kumori component build helloworld
...
$ kumori component register helloworld
...
```

3.7.2. Update and register a new version of Hello World service

The next step is to register a new version of the service application using the updated component instead of the old one. We do that by assigning the new version of FE component to the service `helloworld-fe` role.

Edit the file `services/YOUR_CHOSEN_DOMAIN/helloworld/Manifest.json` and change the service name and component version to `0_0_2`:

```
{
  "spec": "http://eslap.cloud/manifest/service/1_0_0",
  "name": "eslap://YOUR_CHOSEN_DOMAIN/services/helloworld/0_0_2",
  ...
  "roles": [{
    "name": "helloworld-fe",
    "component": "eslap://YOUR_CHOSEN_DOMAIN/components/helloworld/0_0_2"
  }],
  ...
}
```

The new service application is registered as we did previously in section [Register Hello World service application](#).

```
$ kumori service register helloworld
```

3.7.3. Deploy a new service

The next step is to deploy a new instance of `helloworld` service application but, this time, using the new version `0_0_2`. There are two ways of doing this:

- Update the current deployment manifest we created previously in section [Configure Hello World service application](#) to point to the new version of the service application.
- Create a new deployment manifest, keeping the old one as it is.

For this example we will modify the existing deployment manifest.



Adding a new deployment is as easy as using the `kumori deployment add` command.

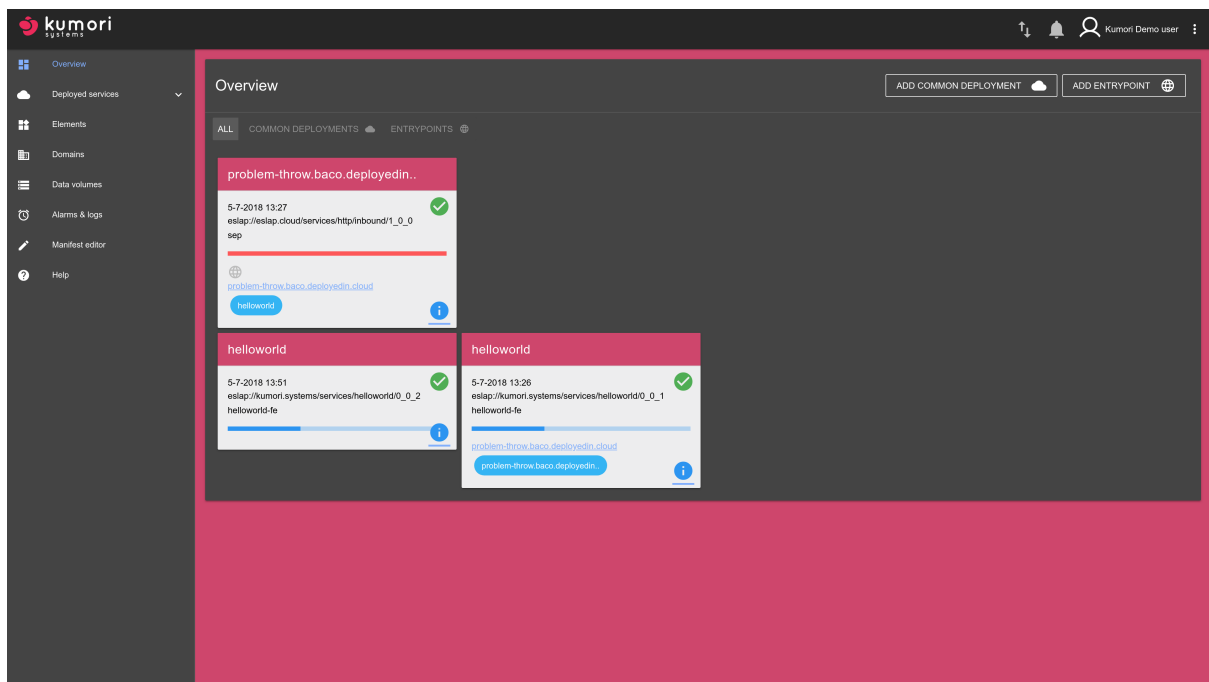
Edit the file `deployments/helloworld/Manifest.json` and change the version of the service application name in `servicename` key:

```
{
  "spec": "http://eslap.cloud/manifest/deployment/1_0_0",
  "servicename": "eslap://YOUR_CHOSEN_DOMAIN/services/helloworld/0_0_2",
  ...
}
```

And that's it!! Now, we can deploy the new version of the service as we did in section [Deploy Hello World service](#).

```
$ kumori deployment deploy helloworld
```

As a result, a new service appears in the dashborad overview page.




3.7.4. Change the endpoint link to the new service

After the previous step, we will have a new service deployed but the endpoint we created in section [Deploy Hello World service](#) will still point to the previous service. If you take a look again to the overview page, notice that the link is still assigned to the `eslap://YOUR_CHOSEN_DOMAIN/services/helloworld/0_0_1` service. Again, we can take two different paths:

- Create a new endpoint and link it to the new service.
- Unlink the existing endpoint from the old service and link it to the new one.

We will take the second approach to keep the same domain.

Unlinking an endpoint is easy. We only have to select the old helloworld service by clicking on its blue Info button  in the overview section. We will get the detailed page of that service. Then, we click the cross icon in the domain assigned to *service* in the *Connections* section

 (note that you will see your random domain instead of `problem-throw`).

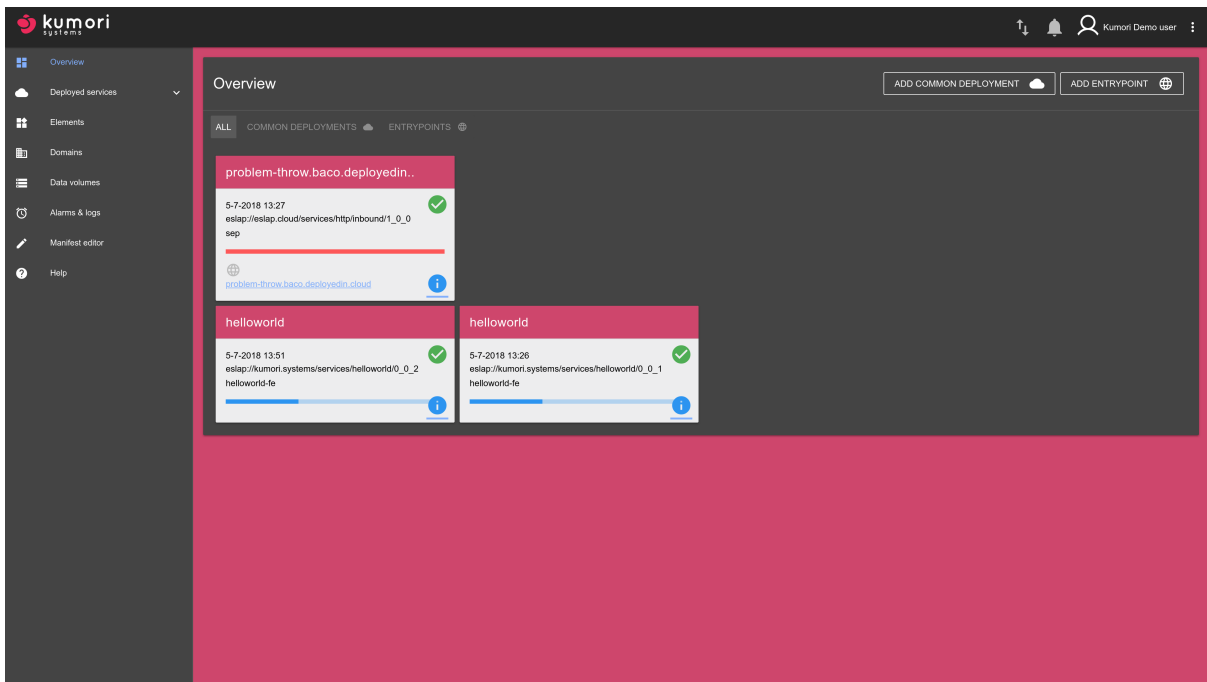
The screenshot shows the Kumori Systems dashboard for a service named 'helloworld'. The interface includes a sidebar with navigation options like Overview, Deployed services, Elements, Domains, Data volumes, Alarms & logs, Manifest editor, and Help. The main content area displays the service details, including its URN, date, and service name. The 'Connections' section shows a connection between the service and a frontend. The 'Role: helloworld-fe' section shows the component, runtime, and channels (entrypoint and service). A graph displays performance metrics (CPU, memory, bandwidth_input, bandwidth_output) over time.

APPLY CHANGES

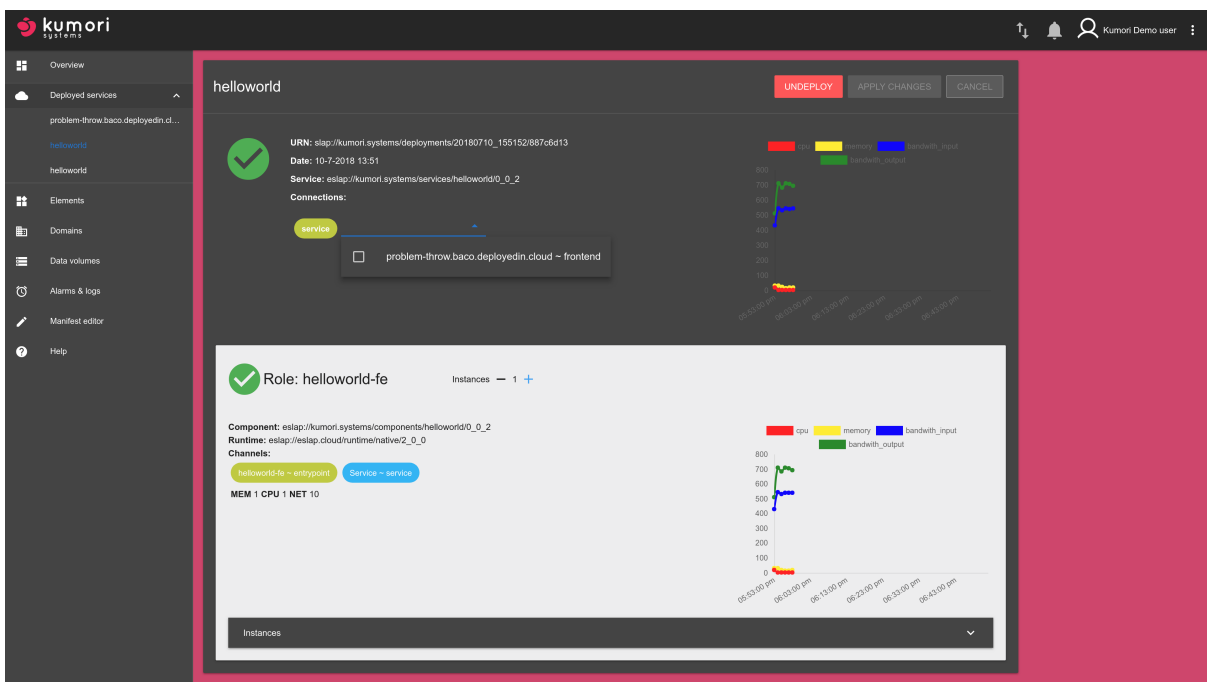
And apply changes

This screenshot is similar to the first one, but the 'Connections' section now shows the service connected to 'entrypoint' instead of the frontend. The 'Role: helloworld-fe' section remains the same, showing the component, runtime, and channels. The graph also displays performance metrics over time.

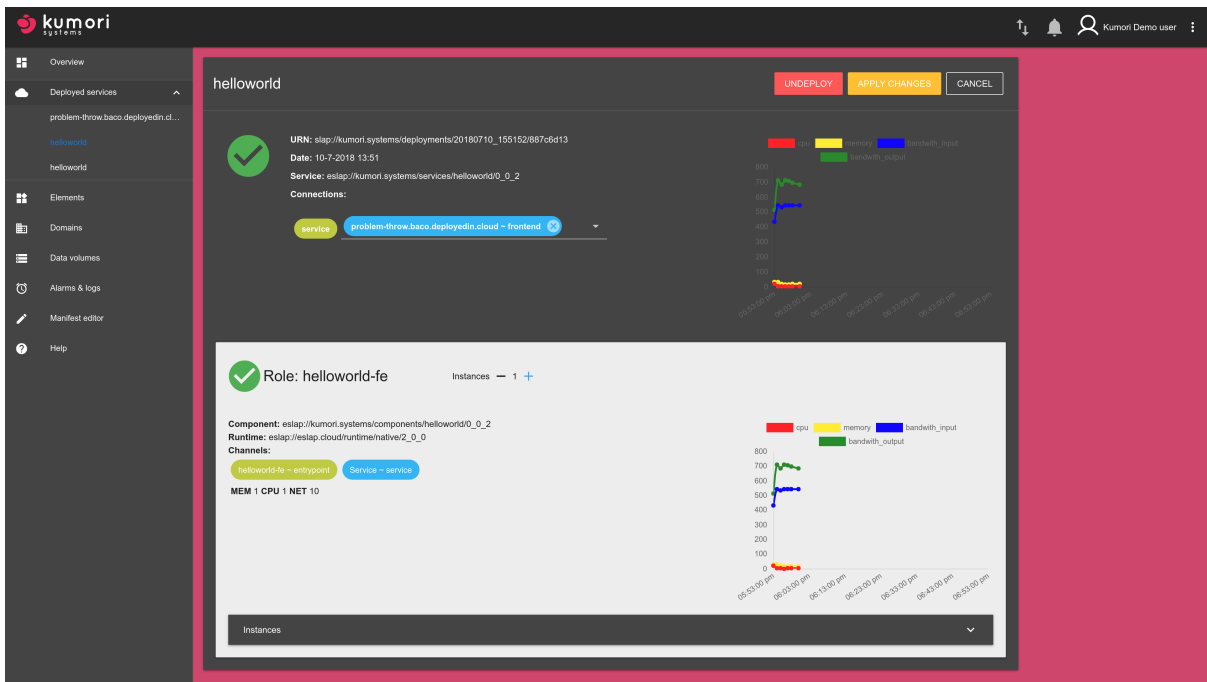
If we go back to the overview page, we will see now that none of the existing services have the entrypoint assigned.



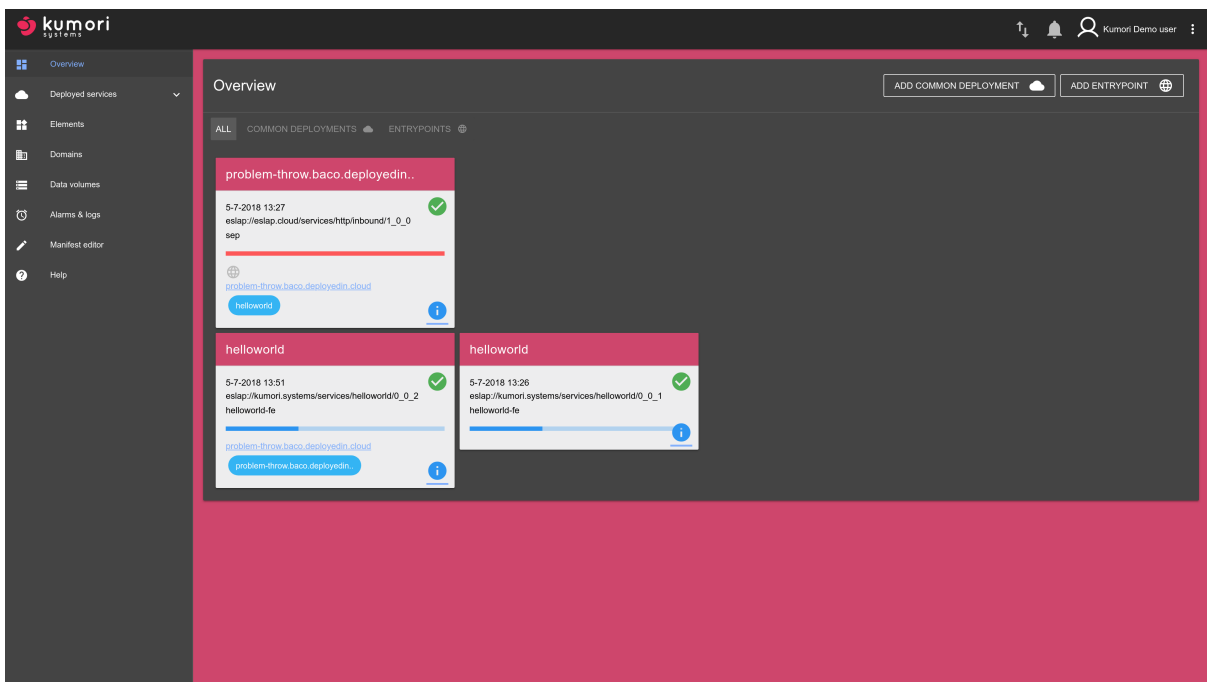
Click not the Info button of the `eslap://YOUR_CHOSEN_DOMAIN/services/helloworld/0_0_2` service and select the inbound service in the service channel drop down list of the Connections section.



And apply changes APPLY CHANGES.



If you go back again to the overview page, the endpoint is now assigned to the new service,



Check the link with the `/api/sayhello` and you will get **Hello world updated!**.

3.7.5. Remove the old service

Once we checked that everything is working, we can remove the old service. We can do that either from the Kumori CLI or through the dashboard itself by clicking clicking on its blue Info

button  of the `eslap://YOUR_CHOSEN_DOMAIN/services/helloworld/0_0_1` and, then,

the *Undeploy* button . We will use CLI in this example:

```
$ kumori deployment undeploy slap://kumori.systems/deployments/20180702_150513/ed8a7c2e
```

4. The Kumori PaaS Service Model in detail

As we have briefly seen, manifests are used to describe and declare elements in Kumori PaaS. A manifest includes element type (component, service application, service deployment) and its characteristics (name, configuration parameters, channels, connectors, etc.).

A more detailed description of the elements of the Kumori PaaS Service Model and their associated manifests is provided hereunder.

4.1. Component

A component is a standalone piece of software that can be scaled independently. Although a component can be arbitrarily complex, it is recommended to encapsulate simple and specific functionality, as microservice-based architectures suggest.

Components communicate among them through **channels** and must implement the **Component** interface (described in section [Component interface](#)) in order to be managed by Kumori PaaS. There are implementations of this interface for Node.js and Java.

Registering a component in Kumori PaaS requires of:

- The component binaries (or the code in case of interpreted languages) and their dependencies (`node_modules` for Node.js components).
- A component manifest declaring its characteristics, including:
 - Name of the component. The name is a URN that must follow the format `eslap://YOUR_CHOSEN_DOMAIN/component/COMPONENT_NAME/VERSION`. E.g. `eslap://kumori.systems/component/helloworld/0_0_1`.
 - Name of the component runtime. It must be one of the runtimes registered in the platform. E.g. `eslap://eslap.cloud/runtime/native/2_0_0`.
 - Channels needed to provide its functionality to other components.
 - Required channels to make use of other components functionality.
 - Name and type of the configuration parameters.
 - Name and type of resources it needs (e.g. volatile or persistent volumes).

Channels are objects provided by Kumori PaaS to component instances based on what is stated in their manifest. The semantics of the channels depend on their type:

- Send: allows sending messages and assigning them a topic (`sendChannel.send(message)`).
- Receive: allows receiving messages and subscribe to topics (`receiveChannel.on('message', (message) => {...})`).
- Request: allows issuing a request that expects an asynchronous response using promises (`requestChannel.sendRequest(message).then((response) => {...})`).

- Reply: allows handling requests and replying them asynchronously (`replyChannel.handleRequest = (request) => {...; return promise;}`).
- Duplex: allows to send messages specifying the recipient and receive messages sent by others. It is like a combination of *Send* and *Receive* channels for point to point communication.



Currently, the channels `protocol` is illustrative metadata and is not processed by the platform. However, this field might be used in the future and we recommend to put realistic values, specially for production ready services.

4.1.1. Component interface

As we said, components must implement the `Component` interface, which is provided as a class that component code should extend. The `Component` interface is described below.

```
class Component
  constructor: (
    @runtime,           ①
    @role,             ②
    @iid,              ③
    @incnum,          ④
    @localData,       ⑤
    @resources,       ⑥
    @parameters,     ⑦
    @dependencies,   ⑧
    @offerings       ⑨
  ) ->

  run: () ->          ⑩

  shutdown: () ->    ⑪

  reconfig: (parameters) -> ⑫

module.exports = Component
```

- ① Object providing toolkit API of the runtime agent that runs the component
- ② Name of the role carried out by the component.
- ③ Identifier (string) assigned to the instance of the component. Typically there will be several instances of the component at a given time (variable over time depending on the load and performance).
- ④ Incarnation number. If an instance "dies" unexpectedly (e.g. due to a bug), the platform will restart it and increase this value.
- ⑤ Path where the instance can store data. It is volatile, and data persistence is not ensured on instance restart or relocation. All instances have this resource by default.
- ⑥ Dictionary of assigned Kumori resources to the instance. For example, a persistent volume.
- ⑦ Dictionary containing the values for the parameters declared in the component's manifest. Those values are set during the service deployment process.
- ⑧ Dictionary of the channels required by the component, through which it can issue requests to

other roles or services. Keys are channel names, values are channel objects.

- ⑨ Dictionary of the channels offered by the component, through which it can answer requests from other roles or services.
- ⑩ Method invoked by Kumori PaaS to start instance execution.
- ⑪ Method invoked by Kumori PaaS to warn instance about its imminent shutdown. Instance should take necessary actions in this situation, persisting its state if needed. If the instance doesn't gracefully shutdown, it will be killed.
- ⑫ Method invoked by Kumori PaaS to modify instance configuration.

4.2. Service application

A service application is a set of components interconnected to provide a certain functionality. Each component carries out a **role** and its channels are paired using **connectors**.

A service application also declares its own channels that can be used to link it to other services. A service channel must necessarily be paired with a channel of one of the roles that compose the service application.

A service application is defined in a manifest, the **service manifest**, which declares:

- The service application name. The name is a URN that must follow the format `eslap://YOUR_CHOSEN_DOMAIN/service/SERVICE_NAME/VERSION`. E.g. `eslap://kumori.systems/service/helloworld/0_0_1`.
- The roles that compose the service and which components will carry them out.
- The service parameters, their type and how their values will be propagated to role parameters.
- The resources required for the service, their type and how they are distributed among the roles.
- The connectors that will pair the role channels. There are three possible types of connectors:
 - Publish/Subscribe: allows to pair *Send* and *Receive* channels.
 - Load Balancer: allows to pair *Request* and *Reply* channels.
 - Full: allows to pair *Duplex* channels.

A service registration will fail if its required components have not been also registered before.

4.3. Service

A service is the result of deploying a service application with a certain configuration. The deployment process involves creating instances of each role component and configuring them appropriately. Kumori PaaS can host multiple services of a single service application at the same time, each with its own configuration and component instances. The execution of deployment process starts with a **deployment manifest** that must contain, among other things:

- The name (URN) of the service application to be deployed.
- The value for each configuration parameter declared in the service application manifest.

- The resource elements (e.g. volumes) to be assigned to the service.
- The initial system resource allocation (e.g. CPU and RAM units) per role.

Once deployed, the number of instances assigned to each role in a service will vary over time depending on their workload fluctuation. When an instance sends a message through one of its channels, it will reach one or more instances of the paired channels, depending on the type of connector used:

- Publish/Subscribe: reaches all target instances. If the message includes a topic, only instances subscribed to that topic will receive the message.
- Load Balancer: the request will only be handled by one of the instances.
- Full: the recipient is determined by the instance sending the message.

Since the components are standalone pieces of software, they can be scaled independently. Because Kumori PaaS knows the topology of services and manages communication channels, it is able to understand the interdependencies between roles in a service and take them into account to anticipate possible variations in the environment (e.g. load).

4.4. Manifest versioning

Once an element has been registered in Kumori PaaS (e.g. the component `eslap://kumori.systems/component/helloworld/0_0_1`), it cannot be modified.

Any modification requires registering again the element with an incremented version in its manifest (e.g. `eslap://kumori.systems/component/helloworld/0_0_2`).



If the element is referenced in another manifest, you will need to update the later (and increment its version) as well.

5. Hello World example manifests in detail

There are three manifests in Hello World example: one for the FE component, another for the service application, and a final one for the specific deployment of the service application that we have get into Kumori PaaS.

5.1. FE component

```

{
  "spec": "http://eslap.cloud/manifest/component/1_0_0",
  "name": "eslap://kumori.systems/components/helloworld/0_0_1", ①
  "runtime": "eslap://eslap.cloud/runtime/native/2_0_0", ②
  "code": "helloworld-code-blob", ③
  "configuration": {
    "resources": [ ], ④
    "parameters": [{ ⑤
      "name": "logzioToken",
      "type": "eslap://eslap.cloud/parameter/string/1_0_0"
    }]
  },
  "channels": { ⑥
    "requires": [ ],
    "provides": [{
      "name": "entrypoint",
      "type": "eslap://eslap.cloud/channel/reply/1_0_0",
      "protocol": "eslap://eslap.cloud/protocol/message/http/1_0_0"
    }]
  },
  "profile": {
    "threadability": "*"
  }
}

```

- ① Name of the component.
- ② Name of the component runtime. In this case, the corresponding to Node.js (native runtime of Kumori PaaS).
- ③ Name of the *bundle* that contains all the stuff (e.g. code) belonging to this component.
- ④ Resources it needs (e.g. volatile or persistent volumes). Not used in this example.
- ⑤ Component parameters. In the component instantiation process (Component class constructor), it will be given a value for the `logzioToken` parameter, of *string* type. These values are set in the deployment manifest, and are propagated to all instances of this component.
- ⑥ Component channels. In this case, it provides a single *Reply* channel named `entrypoint`. This channel is not used for communicating to other components (since there are none), and is linked (service manifest) to the service channel.

5.2. Service application

```

{
  "spec": "http://eslap.cloud/manifest/service/1_0_0",
  "name": "eslap://kumori.systems/services/helloworld/0_0_1", ①
  "configuration": {
    "resources": [],
    "parameters": [{ ②
      "name": "helloworld-fe",
      "type": "eslap://eslap.cloud/parameter/json/1_0_0"
    }]
  },
  "roles": [{ ③
    "name": "helloworld-fe",
    "component": "eslap://kumori.systems/components/helloworld/0_0_1"
  }],
  "channels": { ④
    "provides": [{
      "name": "service",
      "type": "eslap://eslap.cloud/channel/reply/1_0_0",
      "protocol": "eslap://eslap.cloud/protocol/message/http/1_0_0"
    }],
    "requires": []
  },
  "connectors": [{ ⑤
    "type": "eslap://eslap.cloud/connector/loadbalancer/1_0_0",
    "depended": [{
      "endpoint": "service"
    }],
    "provided": [{
      "role": "helloworld-fe",
      "endpoint": "entrypoint"
    }]
  }]
}

```

- ① Name of the service application.
- ② Spreading of the parameter values from the ones set in the deployment manifest to the parameters declared in the component.
- ③ List of *roles* that compose the service application. A component can carry out one or more roles in the service, and instances will be created for each of them. In this example, the component `eslap://kumori.systems/components/helloworld/0_0_1` is assigned a single *role* named "helloworld-fe".
- ④ Service channels. In this case it provides a single *Reply* channel named `service`, which allows access from the outside to the REST API that provides the service.
- ⑤ Connectors. We use a *LB* connector to link the `service` service channel to the `entrypoint` channel of the FE component. *LB* connectors pair *Request* channels with *Reply* channels, with one exception: when they connect a service *Reply* channel to a component *Reply* channel (in this case, *LB* performs *forwarding* functions).

5.3. Service application deployment

```

{
  "spec": "http://eslap.cloud/manifest/deployment/1_0_0",
  "servicename": "eslap://kumori.systems/services/helloworld/0_0_1", ①
  "name": "helloworld", ②
  "configuration": {
    "resources": {},
    "parameters": { ③
      "helloworld-fe": {
        "logzioToken": "xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx"
      }
    }
  },
  "roles": {
    "helloworld-fe": {
      "resources": { ④
        "__instances": 1,
        "__maxinstances": 3,
        "__cpu": 1,
        "__memory": 1,
        "__ioperf": 1,
        "__iopsintensive": false,
        "__bandwidth": 10,
        "__resilience": 1
      }
    }
  }
}

```

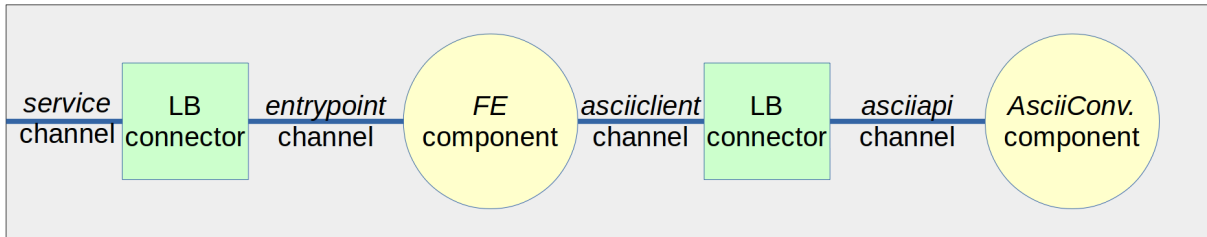
- ① URN of the service application to be deployed.
- ② Short name for the service. You will see this name in Kumori Dashboard, for example.
- ③ Initial values for the component parameters. The various possibilities of the spreading mechanism from the values set in the deployment to the parameters declared in the components are beyond the scope of this guide.
In this case, we use a very simple mechanism: in the deployment manifest we have a *json* for each of the components, the values of this *json* are mapped one-to-one with the role parameters.
- ④ For each role, we set the initial number of instances, the maximum number of instances (Kumori PaaS won't scale the role beyond this amount), resource allocation units, and resilience.

6. Hello World v2: FE + AsciiConverter (with Load Balancing)

The second example is a service application to convert images to an ASCII character matrix. This service can be accessed through a Single Page Application (SPA) web page.

As in the previous example, we have a FE component that implements the REST API using Express. The image conversion falls upon a second component: AsciiConverter.

The communication between both components is of Request-Reply type, so both are connected through an *LB* connector (FE using *Request* channel, AsciiConverter using *Reply* channel).



If the replier component holds a state, it **will not** be shared across all its instances. If there is more than one instance during the life cycle of the service, then a request might be handled by an instance that does not have access to the state shard required to solve it.

6.1. Populate your workspace with Hello World V2 sample service

For this second example, we are going to use the `@kumori/workspace:project-hello-world-v2` template, that provides the fully implemented FE and Asciiart Converter components and the service manifest describing the service topology. We are going to name the project as `helloworld_v2`:

```
$ kumori project add -t @kumori/workspace:project-hello-world-v2 helloworld_v2
```

This will create the following elements:

- A component under `components/YOUR_CHOSEN_DOMAIN/helloworld_v2_fe/` workspace path.
- A component under `components/YOUR_CHOSEN_DOMAIN/helloworld_v2_ascii/` workspace path.
- A service under `services/YOUR_CHOSEN_DOMAIN/helloworld_v2/` workspace path.
- A deployment manifest under `deployments/helloworld_v2/` workspace path.

6.1.1. The component `helloworld_v2_fe`



The source code of this example is extensively commented and we encourage you to review it for implementation details.

The implementation of the FE component is very similar to that of the first example. However, in this case the RestAPI delegates the processing of the request to the AsciiConverter component via the `asciiclient Request` channel.

The method `RestAPI._processRequest()` shows how to send a message through a `_Request_` channel and receive the reply asynchronously through a promise.

6.1.2. The component `helloworld_v2_ascii`



The source code of this example is extensively commented and we encourage you to review it for implementation details.

AsciiConverter converts images to ASCII.

Through the `asciiapi Reply` channel, it receives requests from the FE component.

The method `AsciiConverter._handleRequest()` shows how to process messages received by a `_Reply_channel`.

6.1.3. The `helloworld_v2` service

This time declares two roles:

- `helloworld_v2_ascii`: converts images to ascii.
- `helloworld_v2_fe`: serves de SPA and redirects REST calls to `helloworld_v2_ascii`.

The topology is also extended with a new connector linking `asciiclient` channel from `helloworld_v2_fe` to `asciiapi` channel from `helloworld_v2_ascii`.

6.1.4. The `helloworld` deployment manifest

This time the deployment configuration is used to configure both components logger. See section [Manifests](#) for detailed information about this example manifests.

6.2. Configure logging system (set `logzioToken` parameter)



The following is not mandatory (our examples work), but we encourage you to do it, especially if you introduce modifications in the code.

As in the first example, set the `logzioToken` value in `deployments/helloworld-v2/Manifest.json`.

You will be able to check logs in <https://app.logz.io/#/dashboard/kibana> once you have deployed the service.

6.3. Manifests

We have four manifests: one for each of the components (FE and AsciiConverter), another for the service application, and a final one for the service application deployment.

6.3.1. FE component

```

{
  "spec": "http://eslap.cloud/manifest/component/1_0_0",
  "name": "eslap://kumori.systems/components/helloworld_v2_fe/0_0_1",
  "runtime": "eslap://eslap.cloud/runtime/native/2_0_0",
  "code": "helloworld_v2_fe-code-blob",
  "configuration": {
    "resources": [ ],
    "parameters": [{
      "name": "logzioToken",
      "type": "eslap://eslap.cloud/parameter/string/1_0_0"
    }]
  },
  "channels": {
    "requires": [{ ①
      "name": "asciiclient",
      "type": "eslap://eslap.cloud/channel/request/1_0_0",
      "protocol": "TBD"
    }],
    "provides": [{
      "name": "entrypoint",
      "type": "eslap://eslap.cloud/channel/reply/1_0_0",
      "protocol": "eslap://eslap.cloud/protocol/message/http/1_0_0"
    }]
  },
  "profile": {
    "threadability": "*"
  }
}

```

- ① The FE component manifest is basically the same as in the first example, to which we have added a *Request* channel. It appears in the `channels/requires` section as it is a dependency of the FE component on another component.

6.3.2. AsciiConverter component

```

{
  "spec": "http://eslap.cloud/manifest/component/1_0_0",
  "name": "eslap://kumori.systems/components/helloworld_v2_ascii/0_0_1",
  "runtime": "eslap://eslap.cloud/runtime/native/2_0_0",
  "code": "helloworld_v2_ascii-code-blob",
  "configuration": {
    "resources": [ ],
    "parameters": [{
      "name": "logzioToken",
      "type": "eslap://eslap.cloud/parameter/string/1_0_0"
    }]
  },
  "channels": {
    "provides": [{ ①
      "name": "asciiapi",
      "type": "eslap://eslap.cloud/channel/reply/1_0_0",
      "protocol": "TBD"
    }],
    "requires": []
  },
  "profile": {
    "threadability": "*"
  }
}

```


- ① The AsciiConverter component provides (channels/provides) a Reply channel.

6.3.3. Service application

```
{
  "spec": "http://eslap.cloud/manifest/service/1_0_0",
  "name": "eslap://kumori.systems/services/helloworld_v2/0_0_1",
  "configuration": {
    "resources": [],
    "parameters": [{
      "name": "helloworld_v2-fe",
      "type": "eslap://eslap.cloud/parameter/json/1_0_0"
    }],
    [
      {
        "name": "helloworld_v2-ascii", ①
        "type": "eslap://eslap.cloud/parameter/json/1_0_0"
      }
    ]
  ],
  "roles": [{ ②
    "name": "helloworld_v2-fe",
    "component": "eslap://kumori.systems/components/helloworld_v2_fe/0_0_1"
  }, {
    "name": "helloworld_v2-ascii",
    "component": "eslap://kumori.systems/components/helloworld_v2_ascii/0_0_1"
  }],
  "channels": { ③
    "provides": [{
      "name": "service",
      "type": "eslap://eslap.cloud/channel/reply/1_0_0",
      "protocol": "eslap://eslap.cloud/protocol/message/http/1_0_0"
    }],
    "requires": []
  },
  "connectors": [{
    "type": "eslap://eslap.cloud/connector/loadbalancer/1_0_0",
    "depended": [{
      "endpoint": "service"
    }],
    "provided": [{
      "role": "helloworld_v2-fe",
      "endpoint": "entrypoint"
    }
  ]
  }, {
    "type": "eslap://eslap.cloud/connector/loadbalancer/1_0_0", ④
    "depended": [{
      "role": "helloworld_v2-fe",
      "endpoint": "asciiclient"
    }],
    "provided": [{
      "role": "helloworld_v2-ascii",
      "endpoint": "asciiapi"
    }
  ]
  }
}
```

- ① AsciiConverter role also needs to receive the `logzioToken` parameter.
- ② We have added a new role, associated with the AsciiConverter component.
- ③ The service channels section has not changed: the service still has a single endpoint.

- ④ Includes a new *LB* connector to link the *FE Request* channel to the *AsciiConverter Reply* channel.

6.3.4. Service application deployment

```
{
  "spec": "http://eslap.cloud/manifest/deployment/1_0_0",
  "servicename": "eslap://kumori.systems/services/helloworld_v2/0_0_1",
  "name": "helloworld_v2",
  "configuration": {
    "resources": {},
    "parameters": {
      "helloworld_v2-fe": {
        "logzioToken": "xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx"
      },
      "helloworld_v2-ascii": { ①
        "logzioToken": "xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx"
      }
    }
  },
  "roles": {
    "helloworld_v2-fe": {
      "resources": {
        "__instances": 1,
        "__maxinstances": 3,
        "__cpu": 1,
        "__memory": 1,
        "__ioperf": 1,
        "__iopsintensive": false,
        "__bandwidth": 10,
        "__resilience": 1
      },
    },
    "helloworld_v2-ascii": { ②
      "resources": {
        "__instances": 1,
        "__maxinstances": 3,
        "__cpu": 1,
        "__memory": 1,
        "__ioperf": 1,
        "__iopsintensive": false,
        "__bandwidth": 10,
        "__resilience": 1
      }
    }
  }
}
```

- ① AsciiConverter role also needs to receive the `logzioToken` parameter.
- ② We have added a new section of initial resource allocation for the new AsciiConverter role.

6.4. Deploy Hello World v2 service

It's time to deploy! Once again, you can do it using Kumori CLI. However, this time we use the following special flags:

- `--build-components`: builds the required components as long as they have not been previously build or registered.

- `--generate-inbounds`: creates and links an inbound endpoint service to each service channel. For each new inbound, a random subdomain under `deployed.in.cloud` is also created.

```
$ kumori deployment deploy --build-components --generate-inbounds helloworld_v2
```



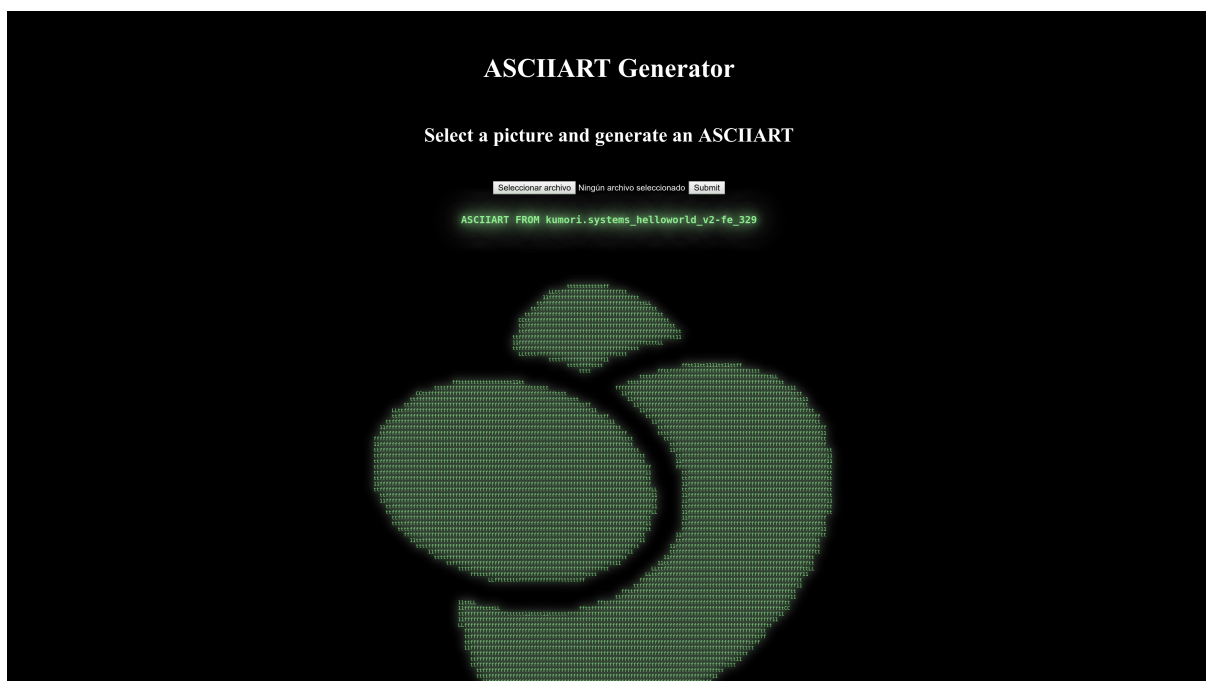
As you may have noticed, we have skipped the build, registration and endpoint creation steps. As we have previously said, `--build-components` and `--generate-inbounds` flags are used to automatically build the components and create the endpoints. The `kumori deployment deploy` command also registers any element required but not previously registered in the platform, as long as it exists in the current workspace and have been already built.

As a result, this time the deployment process will respond with two new services:

- An instance of `helloworld-v2` service. It will include a `URN` key with the new deployment name. Note down this URN since we will use it later in section [Scaling a role in a service](#).
- An instance of an inbound service. The information provided for this service will include a `link` key with the randomly generated URL. This is the URL you will use later in section [Testing the service](#) to test your service.

6.5. Testing the service

We can test the deployed service application by accessing the generated random domain. That will serve a Single Page Application to convert images to ascii:




Select an image and see the resulting ASCII version. Note that the service shows which instance attended your request.

6.6. Scaling a role in a service

We can manually scale any role using the Kumori CLI. As an example, we scale the frontend:

```
$ kumori deployment scale <DEPLOYMENT_URN> helloworld_v2-fe 2
```



you can also manually scale a role from the service detailed view in the dashboard. Click the  icon of your service in the main page (Overview).

Try to convert several images and check if the replier instance id changes.